ADAPT

IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Basic Services Architecture: Month 6 Report

Deliverable Identifier:	D1
Delivery Date:	03/21/2003
Classification:	Public Circulation
Authors:	Özalp Babaoğlu, Alberto Bartoli, Ricardo Jiménez-Peris, Bettina Kemme, Vance Maverick, Marta Patiño-Martínez, Milan Prica, Jakša Vučković and Huaigu Wu
Document version:	1.0, 03/21/2003
Contract Start Date:	1 September 2002
Duration:	36 months
Project Coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd (UK)

Project funded by the European Commission under the Information Society Technology Programme of the 5th Framework (1998-2002)

ist information society technologies

Contents

1 Introduction

This is the 6-month report on the architecture of basic services (BS) in ADAPT. We have chosen to base the architecture on J2EE [Sha02], and specifically on the JBoss open-source implementation [JBo]. This decision is based on investigations in several related directions. We have examined JBoss in detail—not only its documentation, but also its codebase. We have considered the general problem of J2EE replication in the light of the literature on replication techniques. And we have tested several available toolkits for group communication, a fundamental building block of replication.

As a result of these investigations, we have decided that the best way for us to implement basic services is to build on the JBoss implementation of J2EE. However, we will replace its replication mechanism with a new one of our own (probably reusing a few pieces of the replication code). At the same time, we have developed a proposal for database replication middleware, which is essentially independent of the other architectural decisions. And the composite services (CS) architecture of ADAPT has introduced several "glue" components, which are closely associated with the BS architecture.

This document is organized as follows. Section 2 is a critical review of the current state of replication in JBoss. In Section 3, we describe our investigations of some group-communication toolkits. We highlight issues with JavaGroups [Jav], the toolkit used for clustering in JBoss. In Section 4, we discuss the J2EE replication problem in general terms; suggest the range of implementation possibilities; and describe our design and implementation strategy for the next phase of development. In Section 5, we present the database replication middleware. Finally, in Section 6, we describe two "glue" elements of the CS architecture: the "Service Enactment Coordinator", which integrates service invocations into an overall composite process, and the "sensor/actuator" API, which exposes local properties of the basic service installation.

2 Analysis of JBoss Replication

This is an analysis of replication, or clustering, in JBoss version 3.0.6, the latest release at the time of writing.

2.1 Introduction

When clustering a J2EE environment we need to consider various problems:

Replicating resources Resources such as EJBs, the JNDI tree etc are replicated on various nodes, and their state has to be consistent among the cluster, in the sense that a client must see the same state whichever node it connects to.

Communication Application components deployed on different nodes have to be able to communicate as if they were in the same JVM. In J2EE this is achieved with RMI.

Concurrency control Concurrent accesses to resources must be handled correctly. With a few exceptions, EJBs do not support concurrent access. The EJB Specification defines a number of rules to follow when dealing with concurrent accesses to the same bean. This behavior must be provided also on a cluster wide basis.

Failure handling If there is a failure while processing an operation, in some cases the operation must be re-executed or resumed on another node transparently to the client. We will see how different failures are handled.

Controlling accesses to External Resources Re-execution of requests must not produce multiple accesses to external resources such as messaging systems, data storage or legacy systems.

Transactions The behavior of transactional objects must be the same as in a non-replicated system. If a method invocation fails over to a replica, for example, the invocation in the replica must be in an equivalent transactional context.

When talking about replication we need to distinguish between replicating components of the application (EJB classes, Servlet classes etc.), and replicating run-time data such as instances of these components. In the former case components just need be deployed cluster wide. In JBoss there is a mechanism that automatically replicates components deployed on one node to the others. This mechanism is called Farming.

From now on, if not stated otherwise explicitly, we will refer to replication of run-time data. So with the term EJB we indicate a specific instance of a bean. For replicating run-time data there are basically four types of data structures that need to be replicated:

- Stateful Session Beans
- Entity Beans
- HTTP Session objects
- The JNDI tree

Notice that instances of Stateless Session Beans need not be replicated, but only deployed on the whole cluster.

Before going in deep with the replication mechanisms of these structures we have to spend a few words on the configuration of a JBoss cluster. We call a cluster a set of interconnected JBoss instances usually running on different computers. Each instance is indicated as a node. The cluster can be divided into partitions.[1] Each component is associated with a partition, so when it gets deployed it doesn't get replicated to the whole cluster but only on its partition. A node belongs to one or more partitions. This means that partitions can overlap, i.e. multiple partitions can share some nodes.

2.1.1 Subpartitions

We have put into evidence the difference between component classes and component instances, now we will do a similar thing with replication. Replication in general has two purposes: providing fault tolerance and enabling distribution of requests to a number of servers under high load. When we bring this vision in the Stateful Session Bean (SFSB) context we can identify two respective replication forms. Since each SFSB instance is associated only with one client we need only to replicate that specific SFSB instance for providing fault tolerance.

On the other side the session bean's class has to be deployed on many servers so we can distribute the clients on these servers. Often replication for load distribution has different needs for the number of replicas from the fault tolerance replication. For example we could have the bean deployed on 12 servers, in order to be able to handle a large number of clients, but each bean instance would be replicated only on 3 servers, because we do not need more fault tolerance than that. Now the total load of the cluster is distributed among 12 servers, but if a node fails there are only 2 other servers that hold the same SFSB instances and that can take its place. In this way we can keep low the number of state transfers between servers, and provide some form of scalability.

This concept is being brought into practice in JBoss through the use of subpartitions. As we said a JBoss cluster is divided into partitions. A SFSB gets deployed on the whole partition it is associated with. Partitions are further divided into subpartitions. But each instance of the SFSB is replicated only on the subpartition where it got instantiated. In other words partition size is chosen for load distribution, subpartition size is chosen for fault-tolerance.

2.2 Stateful Session Bean Replication

Stateful Session Bean Replication is achieved by multicasting the SFSB state after each invocation to the bean. This broadcast is synchronous (blocking) in the sense that the sender waits for an acknowledgement from each receiver before responding to the client. We will examine in a moment how node failures are handled.

In JBoss version 3.0.6, a distributed lock mechanism was introduced to prevent concurrent access to the same SFSB. Unfortunately locks are still used in an incorrect way. Instead of acquiring locks before the invocation of the bean, they are acquired only before the state transfer of the bean, after the invocation. Moreover locks are released just after the state transfer, while they should be held until the clients request is processed completely.

[1]Notice that these partitions are not the same as "network partitions" that are essentially failures in the communication infrastructure, but a standard configuration mechanism used in JBoss. When we talk about failures we will indicate explicitly that we are talking about network partitions.

However this problem is avoided in the recommended JBoss configuration which is to use homogeneous deployment with sticky HTTP session. This means that each component is deployed on each server instance and the same client always connects to the same server instance in the same session. Being the Session Beans associated with the client session, we have the property that a Session Bean instance will be always accessed on the same server instance. In this way, in absence of failures, we do not have to deal with concurrent accesses. This also relaxes the requirements of the underlying group communication layer. In fact we do not need total ordering broadcast semantics, because only one node broadcasts the state update messages of a specific SFSB instance. Moreover, the fact the broadcast is synchronous (blocking) guarantees us that the node processing requests from a specific client will not send any other state broadcasts, regarding that same bean, before the other nodes have received the update.

2.2.1 Stateful Session Bean State

When the state update message is broadcast to the other nodes, the message contains the whole state of the bean not just a difference from the previous state. The EJB Specification defines the conversational state of SFSBs to be the serialized form of the bean instance plus any open resources held by the bean. It is up to the bean developer to acquire and release these resources with the ejbActivate() and ejbPassivate() methods.

JBoss in fact uses java serialization to obtain a bean's state. Before serializing a bean, the ejbPassivate() method is called and after the serialization is done the ejbActivate() method is called.

2.2.2 Failures

When using sticky sessions, a client could change the server instance it uses only when there is a failure of that server. Various types of failures are treated differently. In general the client stub can transparently fail over to another server or raise an exception to the client application. JBoss generates "intelligent" client stubs that are aware of the server replication and encapsulate the clustering policy.

The general behavior is that a client stub fails over to another server if there is a communication error, this includes also the case when the server crashes or is unreachable. On the other side, if the server reports a GenericClusteringException for some reason, this exception also includes a status flag that indicates the completion of the request. The flag can have three values:

COMPLETED_YES The request has been processed by the server node

COMPLETED_NO The request has not been processed and it can be retransmitted to another node

COMPLETED_MAYBE We do not know if the request has been processed

The client fails over to another node only in the case the GenericClusteringException reported the completion code COMPLETED_NO.

For example if the server crashes between the executions of two requests, when the client tries to contact the server it gets no response. In this case the client stub simply sends the same request to another node. No exception is reported to the client application, the stub automatically contacts another server instance.

On the other hand, if the server receives the request but crashes while processing it, the client contacts another node. In this case the old crashed server could have broadcast the updated state and the new server could have already received the state or could have a pending update when it receives the client's fail-over request. For this reason the client includes along with the other data a flag in the invocation message that tells if this is a failover invocation or the first one. In this way the server can know whether to process the clients request or not.

A delicate point for failures is during the broadcast phase when the sender is waiting for the acknowledgement of all receivers. If one of the receivers fails the sender could block for ever waiting for its response. But on the other side if a new node joins the cluster during this phase, the sender must not expect an acknowledgement from this node, since the node never received the request. In order to deal with this, JBoss takes a snapshot of the current cluster view when doing the broadcast and then it waits for acknowledgements only from the members of that snapshot, excluding any nodes that join the cluster in the meanwhile. Moreover if the underlying layer reports that a node has failed, it is marked so that it is not expected to send an acknowledgement. The broadcast phase completes when all the nodes from the initial broadcast have responded with an acknowledgement or have been marked as failed.

A bean that receives a request from the client could invoke other beans in order to process that request. In this case JBoss would replicate the state of the other SFSBs each time they get invoked. But if the server fails after having replicated some of the modified beans but before having completed the processing of a clients request, the client would automatically fail over to another node that contains an inconsistent state of the beans.

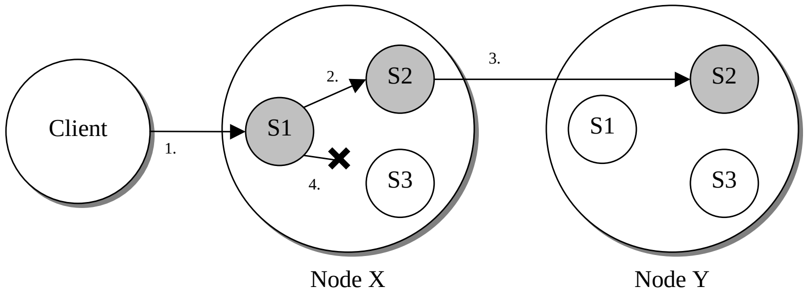

Figure 1: Partial Replication Failure—Client invokes S1, S1 invokes S2, S2 gets replicated then the node X crashes before S3 was invoked.

To clear things out lets look at an example. We have three SFSBs, S1, S2 and S3 replicated on nodes X and Y. Bean S1 on node X receives a request from the client, then it invokes bean S2. The state of S2 gets replicated on node Y. In order to complete the request A must invoke bean S3 also, but node X crashes before. We end up with the situation where the bean S2 has been updated on node Y but not S1 and S3.

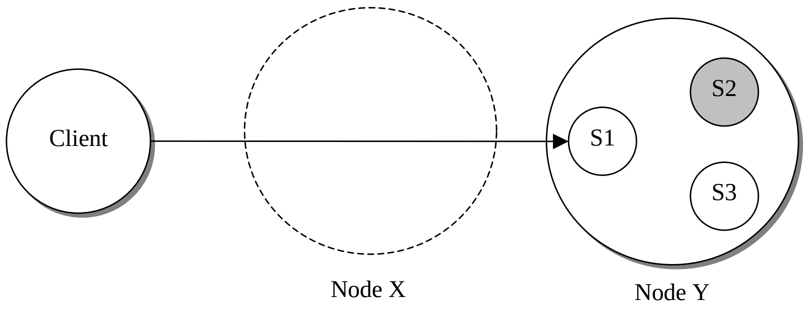

Figure 2: Partial Replication Failure—We end up with an inconsistent state in node Y

We have run tests and in fact it showed that in this scenario the behavior of JBoss is incorrect. This is due to the lack of a replication aware transaction manager in JBoss. The surviving node does not realize that the node which was processing the transaction failed and the state broadcast by that node should be rolled back. We propose three solutions to this problem:

1. Implementation of a replication aware transaction manager, which would roll back the transaction on all nodes if the node that was processing it crashed.
2. Broadcasting the state of all the modified beans to the other nodes only at the commitment of a transaction. This solution would require as a must that the processing of the entire transaction is completely executed on one node, in other words sticky sessions and homogeneous deployment must be forced.[2]
3. The request id and the response should be propagated along with the bean state. If a bean replica that has been updated receives the same request after a failure of the original instance, it returns the response that it received with the state without re-executing. This solution requires a deterministic environment in the sense that executing a request at any moment in the system must produce the same results.

It is possible for a transaction to span several invocations of a stateful session bean. (The same is true of entity beans.) In this case, solution 1 could not be made transparent to the client. Rolling back the transaction might undo more work than just the current method, requiring some of the calling code to be re-executed. We also discuss this transactional issue in another deliverable [JPPM03].

2.2.3 Network Partitions

A special class of failures are network partitions.[3] When the cluster is divided in two or more partitions the nodes cannot communicate and they cannot keep a consistent state between them. In general there are two approaches to deal with network partitions: the primary partition approach and the partitionable approach. With the primary partition only one of the network partitions is active until the cluster merges again, then the state from the primary partition is transferred to the members that were on the non-primary partition. There are many policies for deciding which the primary partition is, but all require that there be only one primary partition.

In a partitionable approach all partitions can process requests, possibly with some restrictions. When the cluster merges, the state in various partitions has to be merged in some way. This state merging is more complicated than the one used for the primary partition approach.

We can say that JBoss uses a partitionable approach with some restrictions when dealing with SFSBs.[4] If using sticky sessions, a client always contacts the same server. If under a network partition that server is unreachable the client changes server until it reaches a server on its own partition. SFSBs are associated only with one client so we know that a SFSB will never be modified on another partition. Moreover each time a client accesses a server, the clients view of the servers list is updated by that server, so the client will see only the servers on its own partitions and if the server it is bound to crashes it will fail over only on a node on the same partition.

[2]This solution was proposed and implemented by Sacha Labourey (JBoss Group) but never committed to the CVS.

[3]In this paragraph we will refer to the network partition with the term partition and not to the JBoss partition used to configure the cluster.

[4]Actually the approach is a hybrid between partitionable and primary partition.

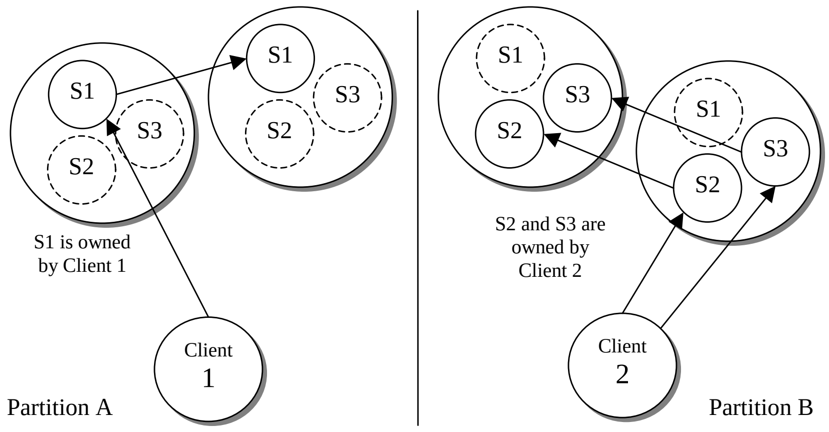

Figure 3: There is a network partition, Client 1 uses partition A and Client 2 uses partition B

When the two partitions merge there is no explicit SFSB state merging. The client doesn't yet know of the presence of the new nodes that joined its partition. Only when the client sends a new request to the node it is bound to, the state of the SFSB will get broadcast on the nodes that have joined the partition and the client will receive the new view of the cluster, so it will be able to fail over also to the other nodes.

This approach works fine in general, but there is a drawback. If the node a client is bound to crashes after a partition merge, but before the client made any requests to the node. The client could not fail-over to the nodes that have joined the partition. Moreover the client must have a different server list for each SFSB. Because if the client has multiple beans on the server, when it calls one of them after a cluster merged, only that bean is replicated on the new members, but the other SFSBs are not up to date on the new nodes. In this the cluster does not provide all the fault tolerance it could provide for a certain period after a merge.

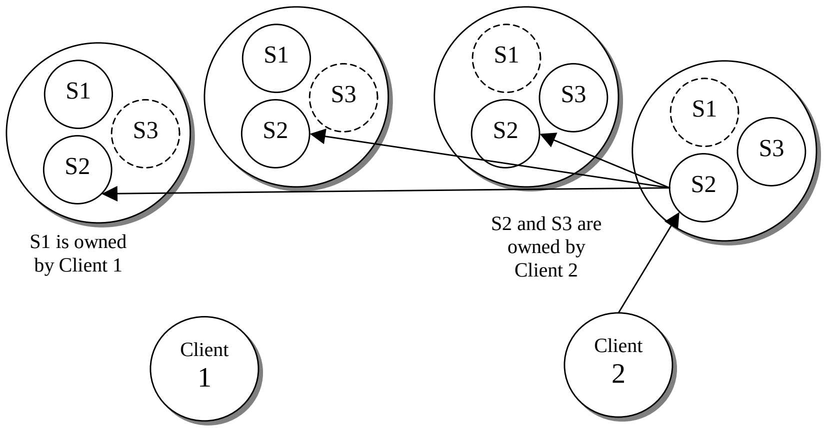

Figure 4: The partitions have merged now, the bean S2 has been replicated on the nodes that have been in partition A, but S3 hasn't yet.

There are still some incorrect behaviors similar to those we've seen when there are crashes with partial request executions. Suppose that there is a network partition and there are two SFSBs associated with our client: S1 and S2. The client calls both of these beans during the network partition. Then the cluster merges and we have S1 and S2 up to date on the nodes that were in our partition, but they are not up to date on the other partition. Now the client invokes the bean S1 and that bean gets replicated to the new nodes. The client receives a new server list for that SFSB. Now the node the client is bound to crashes. The client tries to make another request for S1 but it sees that the node has crashed so it fails over to another node from the list. Suppose it fails over to one of the new nodes that have the up to date state of S1, but the old state of S2. Now if for some reason the processing of the request generates an internal invocation to S2, it would be processed on that node which has an inconsistent state of bean S2. This is another serious problem of the JBoss clustering algorithm.

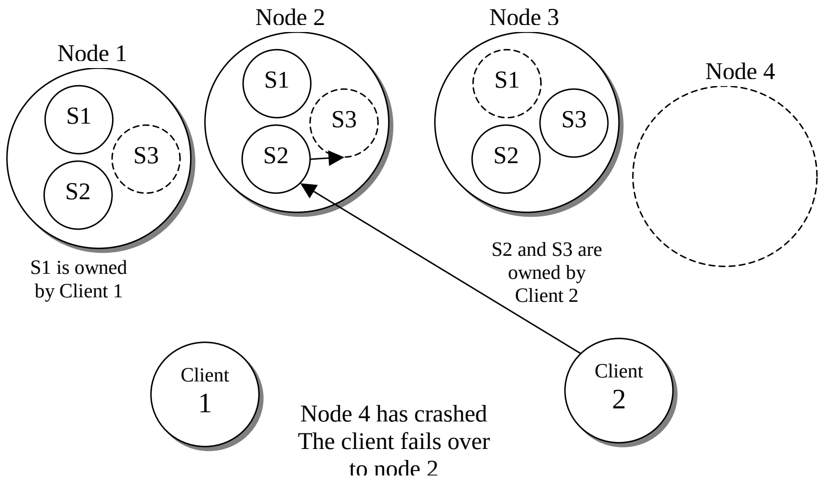
Node 1
S1
S3
S2
Node 2
S1
S3
S2
Node 3
S1
S3
S2
Node 4
S1 is owned by Client 1
S2 and S3 are owned by Client 2
Client 1
Client 2
Node 4 has crashed The client fails over to node 2

Figure 5: The node the client was bound to crashed, the client failed over to a node that has been in the partition A, the client invoked S2, but S2 invoked S3 which is not yet up to date.

This partitionable approach is possible only if the execution of a request does not include Entity Beans or access to the DB. In that case it is up to the DBMS clustering mechanism to define the approach to follow, whether partitionable or primary partition. With a non clustered DBMS the approach is a primary partition approach, where the primary partition is defined to be the one that includes the DB.

2.3 Entity Bean Replication

In JBoss, the state of Entity Beans is not replicated by the Application Server but this task is left to the DBMS. In other words each JBoss server instance accesses the same database and in this way has the same view of persistent data. Entity Beans are loaded from the database before each invocation and stored after the invocation.

As defined in the EJB Specification the Entity Bean developer does not need to worry about controlling concurrent access to an Entity Bean instance, she/he may assume that the container will serialize all accesses to the bean. In other words, both the client and the bean developer must see as if there is no concurrent access to the same Entity Bean instance from different transactions. The EJB Specification also defines three commit options:

Commit Option A The container has exclusive access to the bean state in the persistent storage. It keeps a “ready to use” instance in memory.

Commit Option B The container doesn't have exclusive access to the bean state in the persistent storage, but has to synchronize its state before each transaction. It also keeps a "ready to use" instance in memory

Commit Option C The container doesn't have exclusive access to the bean state in the persistent storage, but has to synchronize its state before each transaction like in option B. It does not keep a "ready to use" instance in memory, instead it returns it each time to the pool of available instances.[5]

Commit option A is not very useful in a clustered environment. As for commit options B and C there are two policies JBoss uses to synchronize access to the bean state in the persistent storage and prevent interferences when two server instances try to concurrently modify the same Entity Bean. These policies are the pessimistic and the optimistic policy.

In the pessimistic policy mutual exclusion is achieved through row locking on the Database where the bean is stored. When an Entity bean is accessed from a transaction, an exclusive lock is acquired for the relative row in the database. This lock blocks any other transactions that are accessing the bean's state. After acquiring the lock, the bean state is loaded from the DB, the business method is called on the bean, and only when the transaction commits, the bean's state is stored to the DB and the lock released. At this point the other blocked transactions can proceed. In this way there will be no two active instances of the same Entity Bean on the whole cluster. This policy guarantees consistency of the database, but has some performance drawbacks because read-only operations are also blocked.

For this purpose JBoss provides an "optimistic" policy. It is possible to define in the JBoss configuration which policy to use. Even if concurrent access to the same bean instance is not permitted by the EJB Specification, the optimistic policy uses a trick to behave like all the transactions were serialized, but actually runs concurrent transactions on different copies of the entity bean. Only one of the concurrent transactions can modify the bean successfully. Before invoking a bean its state is loaded from the DB. When the transaction completes, the container checks if the state of the beans involved in the transaction has been modified in the persistent storage from the time it acquired they were loaded. If not, the new state is stored in the DB.

To explain things better we'll look at an example. Consider we have two transactions A and B that access the same Entity Bean instance. The container does not block either of the transactions because they could be both read-only operations. Let's say the transaction A modifies the bean and finishes first, then the container checks to ensure that the bean state hasn't been changed in the DB and then commits the changes. After that transaction B reaches the end and verifies if the bean state in the DB has changed. It sees that in fact it did change so it rolls back the whole transaction.

With the optimistic policy a system with a lot of update operations would have many roll-back operations. On the other side with a pessimistic policy a system with a lot of read-only operations would block a lot unnecessarily. For these reasons an administrator has to be careful which policy to choose.

As we said concurrent access to an Entity Bean instance from different transactions is never perceived by the bean developer (although the container may do the trick of running concurrent transactions of different copies like in the optimistic policy). But concurrent access to an Entity Bean instance from the same transaction is sometimes possible. By default an Entity Bean in declared as non-reentrant. This means that there can be no callback methods on it. Reentrance is a special case of concurrency. An Entity Bean can be declared as reentrant in its deployment descriptor. In this case it's up to the bean developer to deal with concurrency.

[5]This pool contains empty instances of an Entity Bean class that are not associated with a particular business object instance.

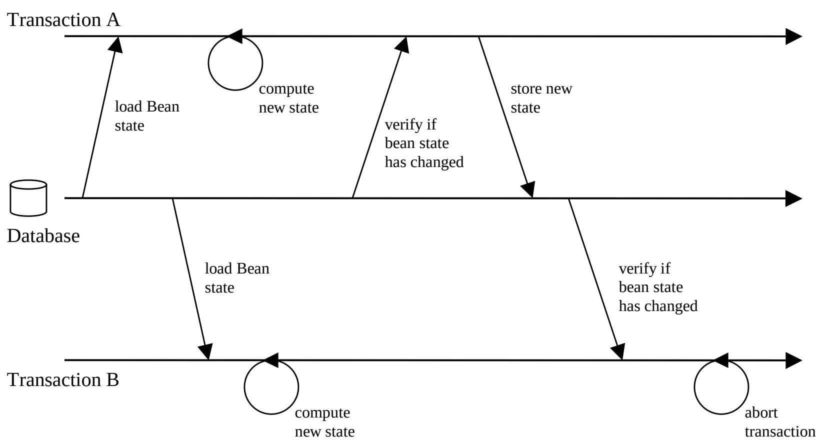

Figure 6: Two concurrent transactions in the Optimistic Policy.

While serialization of accesses to a bean's state in the persistent storage is managed by the container, as we said earlier, entity bean replication is left to the DBMS. In case the DBMS is not replicated we would have a single point of failure for the whole system. It is also left to the DB replication mechanism to decide how to behave in presence of network partitions and handle state merging.

2.3.1 Entity Bean State

When we talk about state, there are two types of Entity Beans: Container Managed Persistence (CMP) and Bean Managed Persistence (BMP).

The state of the CMP Entity Beans is defined using the cmp-fields. These cmp-fields are defined in the deployment descriptor and for each cmp-field there has to be a setter and a getter method. The bean provider has to declare these methods as virtual in the EJB class. The container is responsible for implementing these methods and accessing the persistent storage. The state of each field is defined by the java serialization mechanism.

The state of the BMP Entity Beans is defined by the bean provider through the methods ejbLoad() and ejbStore(). The bean providers responsibility is to synchronize the state of the bean with the DB using these two methods.

2.3.2 Distributed Cache

The advantage of using Entity Beans should be to simulate the behavior of an in memory cache, but this does not happen in JBoss at the moment, since they load and store the state of an Entity Bean on each invocation. Currently a distributed cache for Entity Beans is under development. It is still in an early phase of developement, thus many things haven't been defined yet. The cache will have also distributed locks as a feature. The presence of the distributed cache means that Entity Beans will not have to be loaded from the DB on each invocation.

This distributed cache will be used both for Stateful Session Beans and for Entity Beans. It will be a highly configurable JBoss service, so any other service will be able to use it. At the moment JBoss authors have identified three semantics that the distributed cache will implement.

Asynchronous On each update of the contents the broadcast operation will return immediately.

Synchronous The broadcast operation will wait for acknowledgements.

Serialized Synchronous A distributed lock mechanism will prevent two nodes from modifying the same object concurrently.

It is not yet clear where will the various policies be used and how will the EJB replication algorithm be mapped to the distributed cache.

One of the relevant issues is that the cache will implement the XAResource interface. This means that the state replication on the cluster will be able to take part in a two phase commit protocol along with the other resources involved.

It is probable that the distributed cache will also make possible the replication of the Database in case of container managed persistence. Each node will have a local DB instance and all changes will be replicated on the cluster. So it will not be necessary to use a clustered DBMS in order to replicate those beans. As for the Bean Managed Persistence, the database will be accessed directly by the bean and it won't be replicated automatically by the application server. Of course the database itself could be replicated.

2.4 HTTP Session Replication

The Web Container of JBoss can be either Jetty or Tomcat. Jetty is the default one and it supports all the HTTP Replication strategies:

Extended (JBoss based) This strategy is the default one; it supports the basic replication schemes.

Extended (Jetty based) Session replication based on a Jetty plug-in.

Migratable This strategy doesn't support replication, an application can be moved from one node to another.

Whether Jetty is used or Tomcat, there are three replication policies that define when an HTTP session gets replicated. These are:

Instant Snapshotting Sessions are replicated after each HTTP request.

Interval Snapshotting Sessions are replicated on a time-based interval. This policy is useful if the same client is accessing multiple Servlets concurrently, for example through a HTML frameset. In this case the HTTP session gets replicated only once.

Economic Snapshotting Sessions are replicated only if the setAttribute() method has been called. Notice that the session won't get replicated in case an attribute that is already in the session is modified.

The HTTP Session object gets serialized for replication, thus all objects that are stored in it get serialized as well. The HTTP Session is usually used for storing Stateful Session Bean Instances associated with the owner of the HTTP session. But this doesn't mean that each time we replicate the HTTP Session, all the Stateful Session Beans that are in it get replicated. What's in the HTTP Session object are only references (remote interfaces) to Stateful Session Beans. So when we replicate an HTTP Session only these references get replicated.

The HTTP Session implementation is a serializable object called SerializableHttpSession. The replication of the that object is implemented in JBoss through an Entity Bean. This bean provides a cmp-field called Session, whose setter and getter accept a SerializableHttpSession. The setter is invoked by the servlet container each time it wants to replicate the session. The EJB container that hosts the entity bean replicates the data on the cluster. All information stored in the session gets written to the DB each time we invoke the setter of the Session property of the HTTP Session bean. Synchronization and concurrent accesses are handled by the bean container exactly like for other entity beans.

2.5 JNDI tree replication

The JNDI tree provides a naming, directory and lookup service. Java objects can be stored in the JNDI tree. It is normally used for storing the application configuration environment and finding home interfaces of EJBs. Each EJB is associated with a JNDI name in the tree. When a client wants to create an EJB instance it looks up the Home Interface in the JNDI tree and obtains a stub to the home object of that bean. Then the client invokes the create method through that stub that returns the remote object of the newly created bean instance. Now the client can access all the EJB business methods through this remote object.

In JBoss each node has a local JNDI tree and shares another distributed JNDI tree with the other nodes. Clients that access the JNDI tree are treated differently than EJBs that access the JNDI tree. A client connects to the JNDI service specified in the jndi.properties file. This file contains a list of servers that are hosting the JNDI server. The client tries to connect to a server in the list, and if its not available it tries with the next server from the list. If the list does not contain any addresses, a discovery protocol is used to dynamically find a JNDI server.

When a client looks up an object in the clustered JNDI service, the node contacted by the client first searches in the replicated JNDI tree. If the object cannot be found there it looks into the local JNDI tree. If the object is not even in the local one, then the node asks all other nodes if they have the object in their local JNDI trees. Finally if no node contains the object a NameNotFoundException is raised.

Unlike clients, EJBs access only the local JNDI Tree when they look for objects. This solution was chosen because of compatibility with existing applications. Some applications assumed that the environment was not clustered and EJBs were invoked always on the same machine. Furthermore The EJB container has to keep a separate list for local and remote objects. This solution has also the advantage of keeping a low network traffic with homogeneous clusters.

3 Experiments on group communication platforms under stress

We have carried out several experiments in order to gain insights into the behavior of group communication platforms under stress. By this we mean a scenario in which the application layer (e.g. a J2EE clustering infrastructure) injects a high throughput into the group communication layer (e.g. on the order of 1000 messages per second). As discussed below, our experiments have showed that JavaGroups, the platform used in the JBoss clustering extension, exhibits highly undesirable behavior in such scenarios.

We have analyzed the following platforms, all open-source:

JavaGroups (http://www.javagroups.com) Developed by Bela Ban and used as communication layer for the JBoss clustering extensions. It consists of a number of layers that can be stacked together depending on specific application needs. All layers are written in Java. There is also a version consisting of one single Java layer on top of a non-Java group communication platform (Ensemble, developed at Cornell University). We have analyzed only the full Java implementation because (i) it is the one used by the JBoss clustering extensions; and (ii) Ensemble is a relatively old project.

Spread (http://www.spread.org) Developed at Johns Hopkins University by a team led by Yair Amir. It is used in a number of environments. Spread is implemented in C but a Java programming interface is available. Spread implements the Extended Virtual Synchrony programming model [MAMSA94]. Essentially, this model features totally-ordered delivery and uniform delivery (called *safe* delivery in this model) in a partitionable environment.

JBora JBora is an early prototype developed at the University of Trieste as part of ADAPT. It consists of a Java layer on top of Spread. JBora is meant to export to applications a programming model simpler than Extended Virtual Synchrony while providing total order and uniform delivery in a partitionable environment. More details about JBora will be provided in a later deliverable.

We have configured the three platforms as follows:

JavaGroups Default stack configuration used in the JBoss Clustering extensions (as indicated in [LB02]). Such configuration includes neither total order nor uniform delivery.

Spread Default configuration.

JBora Default configuration for Spread (JBora itself does not need any configuration that could affect its behavior under stress).

We have exercised the three platforms as follows:

JavaGroups FIFO-ordered multicast without uniform delivery. This is the multicast configuration enabled by the stack configuration used in the JBoss Clustering extensions. JavaGroups features a layer that implements totally-ordered delivery, but this is not used in JBoss. The JavaGroups distribution also features a layer that appears to implement uniform delivery, but this layer is qualified as being “obsolete”.

Spread Totally-ordered multicast with and without uniform delivery (called safe delivery in Spread).

JBora Totally-ordered multicast with uniform delivery (called safe delivery in Spread).

Bear in mind that this comparison is in some respects “unfair”:

- The platforms have been exercised with differing ordering and delivery guarantees. Total order is more costly than FIFO order. Obtaining safe delivery is very costly.
- JavaGroups is implemented entirely in Java; Spread is implemented in C and has a thin Java layer on top; JBora is a Java layer on top of Spread.

However, our results are indeed useful to gain insight into the behavior of each system.

3.1 Structure of each experiment

We have considered *throughput* as the only performance index. The code we have used is available upon request and can be customized easily (e.g. for measuring latency or other indices).

There is a predefined number of group members. Each group member is a receiver and some group members are also senders.

Initially, each group member waits to receive a view with the expected number of members. Then, each sender spawns a separate thread for multicasting messages. Finally, each member waits to receive the expected number of messages and then terminates.

Each sender thread executes a loop of *numBursts* iterations. At each iteration it multicasts *msgPerBurst* messages and then sleeps for *nsleep* msecs.

Each receiver determines the local time (`System.currentTimeMillis()`) after receiving the first message and after receiving the last message. Throughput is measured as the number of received messages divided by the difference between these times. The number of received messages will be *numBursts * msgPerBurst * numberOfSenders*.

The code executed by each group member reads the following parameters from a text file:

- Whether the group member is a sender;
- Number of expected group members (4 in all the experiments);
- Message size;
- Number of messages per burst (*msgPerBurst*);
- Sleeping time between bursts (*nsleep*);
- Number of bursts (*numBursts*);
- Number of senders (*numberOfSenders*).

3.2 Operating environment

Software:

- JavaGroups version 2.0.3.
- Spread version 3.16.2
- Windows 2000 Professional SP2

Hardware:

- Four Dell Optiplex GX300 (PIII 800 MHz, 512 MB RAM)
- 100 Mb Switched Ethernet

3.3 Results

Message size is in bytes and times are in msecs. Of course, the workload space has a very large number of dimensions. By no means do the following results provide a complete throughput characterization of the three systems. In particular, we remark again that we focussed on scenarios where the system is highly stressed.

As indicated in the following tables, most of the experiments failed with JavaGroups. This means that at least one of the four processes did not receive all the expected messages, so it did not terminate (see also next section).

Platform	*Senders*	*Msgs/sec*	*KB/sec*
Spread	1	633	316
Spread	2	1279	639
Spread (safe delivery)	1	640	320
Spread (safe delivery)	2	1254	627
JBora	1	576	288
JBora	2	871	435
JavaGroups	1	561	280
JavaGroups **(test failed)**	2	785	392

Test 1: Each sender attempts 1000 msg/sec (10 msgs every 10 msecs): message size 500 bytes.

Platform	*Senders*	*Msgs/sec*	*KB/sec*
Spread	1	316	1583
Spread	2	576	2883
Spread (safe delivery)	1	323	1616
Spread (safe delivery)	2	288	1441
JBora	1	323	1616
JBora	2	359	1717
JavaGroups **(test failed)**	1	316	1583
JavaGroups **(test failed)**	2	275	1379

Test 2: Each sender attempts 1000 msg/sec (5 msgs every 5 msecs): message size 5000 bytes.

3.4 Key highlight

The main highlight is this: *JavaGroups appears unable to sustain large throughput injected into the system*. Some members may *silently* start missing messages, but the other members are not notified of this failure.

In particular, when this phenomenon does occur, the following happens:

1. At least one group member, say *p*, starts missing messages.

Platform	*Senders*	*Msgs/sec*	*KB/sec*
Spread	1	353	3531
Spread	2	373	3737
Spread (safe delivery)	1	359	3590
Spread (safe delivery)	2	365	3651
JBora	1	245	2452
JBora	2	251	2511

Throughput injected by each sender: 4000 msg/sec (20 msgs every 5 msecs); Message size 10000 bytes. JavaGroups seems unable to sustain this throughput.

2. *p* stops receiving messages, i.e. there are no holes in the sequence of messages received by *p*.

3. *p* is not identified as a "faulty" member: the other group members are given no notification at all of the fact that *p* is missing messages. In particular, if the "correct" group members do not leave the group, then *p* will remain a group member "forever"; and if the "correct" group members leave the group, then *p* receives a view change describing the new composition of the group.

While the above behavior is consistent with primary-partition virtual synchrony, it is evident that similar scenarios are highly undesirable in practice.

We do not exclude that the above behavior could be avoided with some parameter tuning of the JavaGroup stack. We have made a few unsuccessful attempts. Nevertheless, we believe that this phenomenon is important to know, in particular, because (i) we have used the *default* configuration for the JBoss clustering extensions; and (ii) it manifests itself only when the system is "stressed".

We have done many tests varying the sleeping time and the message size (keeping the other parameters as in test 2) and we have found that the behavior manifests itself whenever the message size is larger than 2KB. For some combinations of parameters the problem does not always manifest itself. One experiment run may terminates correctly, but a repetition performed immediately afterwards does not terminate.

3.5 Other observations

The performance difference between the three platforms is not "huge" in these experiments. However, experiments with 2 or 3 group members and large throughput showed that JavaGroups performed much worse. Also, we must keep in mind that the ordering and delivery guarantees for JavaGroups are much weaker than those for Spread/JBora.

We expected that the performance of JBora might be lower than that of Spread, because each event received by JBora must pass through a mailbox and a thread context switch. It is not yet clear whether the performance loss that we have observed is important enough to warrant profiling, etc. We'll look at this later.

We have observed that, in each experiment run, Spread/JBora starts delivering messages "immediately", whereas with JavaGroups there is a noticeable startup delay.

3.6 Further experiments with Spread

In order to gain further insights into the behavior of Spread under stress, we have performed additional experiments injecting an even higher load into the system. We have considered the same hardware and software environment as the one previously described. We have performed experiments with one sender that *continuously* injects new multicasts into the system.

We have found that when Spread is unable to sustain the multicasting load injected by the application:

- Spread expels the sending group member from the group. (Note, this is different from virtually partitioning the sending group member from the other members.)
- The surviving group members receive a membership change notifying them about the failure.

In other words, the group member introducing an excessive load is treated as if it had crashed; detection and notification of such reconfiguration is quite fast. (We have not measured this time, which is controlled by the internal timeouts of Spread; however, there is no noticeable delay between the forced failure and the delivery of the view change.) We find this behavior reasonable and much more acceptable in practice than the one we have observed with JavaGroups.

The multicasting load that triggers the above scenario may vary widely between repeated executions. However, we can observe some common patterns. If new multicasts are generated continuously and at full speed, then the sending group member was almost always expelled from the group quickly: after approximately 1 minute for 100-byte messages, and approximately 20 seconds for 1 KB messages. During this time, the number of multicasts injected into the system was a few thousands in the former case and some hundreds in the latter. If new multicasts are generated continuously, but with a sleeping time of 2 msecs between two consecutive multicasts, then the sending group member was almost always expelled from the group quickly, but the time before the forced failure tended to be slightly smaller. We made similar experiments by placing an additional load on the sending machine (a simple application writing on a MySQL database). We observed the same behavior as above, again with a time before the forced failure that tended to be smaller. We remark, however, that these tests were highly demanding because there was a continuous generation of new multicasts.

To summarize, in our tests Spread was able to sustain a continuous load of between 500 and 1000 new multicasts per second depending on the additional load on the sending machine. (Recall that these experiments, like the others with Spread, were performed with the strongest ordering and delivering guarantees.)

We interpret the failure forced by Spread in case of excessive load as due to an overload of the sending buffer. That is, when Spread detects that the amount of data waiting to be multicast is above a certain threshold, Spread expels the group member from the group. This hypothesis is based on our observed comparisons between the multicasts received at the moment of the forced failure with those previously sent. It follows that Spread might forcibly expel a group member even in cases of short peak loads. Clearly, this behavior would be undesirable, in particular because of the many complex software modules that will be executed on a Basic Service machine. We are currently investigating techniques for preventing this scenario from occurring, by applying a form of application-level congestion avoidance. The problem appears to be complex because, according to our observations: (i) the threshold does not depend only on the number of buffered multicasts, but on their aggregate size; and (ii) the threshold appears to be time-varying. That is, we suspend the sending group member when the buffer size is close to the threshold; when the group member is awakened, it appears that Spread has set the threshold to a much lower value.

4 J2EE replication options

We have chosen to base the architecture of ADAPT basic services on Sun's J2EE [Sha02]. In this section, we give a brief overview of J2EE, and discuss how it can be extended to meet the goals of basic services.

4.1 J2EE overview

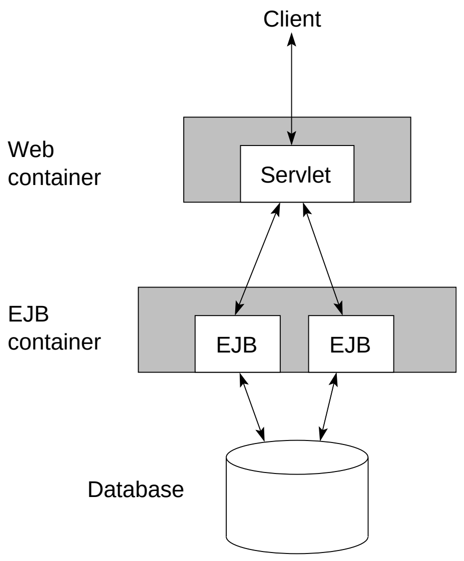

Figure 7: Three-tier architecture of J2EE web service implementation.

As Figure 7 illustrates, J2EE is a three-tier architecture. HTTP requests are handled by “servlets” in the web tier. These invoke business logic encapsulated in “Enterprise Java Beans”, or EJBs. These in turn connect to the database, reading and writing persistent data. The three tiers may be distributed transparently across physical hosts.

The J2EE architecture may be summarized in terms of a variety of *components* and *containers*. A component is a class, or a set of associated classes, written by a developer according to the standard. A container is an engine for running components: it receives the actual request for a component, then instantiates the component and invokes it accordingly. The container may also provide services for the component to call during execution of the request.

For example, in the web tier, the component is a servlet (or a JSP page compiled into a servlet). When an HTTP request arrives, it is handled by the web container, which instantiates the servlet and calls its service method. While the servlet executes, it can read the request, generate the response, and access a state object (the HttpSession) associated with the client-server conversation.

In the EJB tier, the components are session and entity beans. The container instantiates the beans on request. It generally manages the association of transactions with the method invocations (“container-managed transactions”). The EJB container may also automate the mapping of bean state to the database (“container-managed persistence”). Either way, the container supplies the JDBC connection used for reading and writing.

4.2 Asynchronous operations

The J2EE architecture has been developed for client-server and web applications. At both the EJB and web layers, it is designed for request-response operation. This is a primary mode of operation for web services, but by no means the only one. The Web Services Description Language (WSDL) standard [CCMW01] specifies four kinds of operations: in addition to *input-output*, i.e. request-response, operations may be *input-only*, *output-only*, or *output-input*. Further, Alonso et al. [AJB$^+$03] argue that the most important messages for practical web-services use are *asynchronous*, that is, messages whose sender does not expect an immediate response.

In WSDL terms, asynchronous messages correspond to *input-only* and *output-only* operations. Input-only operations can be treated as a special case of input-output operations. The client sends a request, and receives either an empty response message, or an acknowledgment at the transport level. This presents no difficulties for the architecture we have sketched.

With output-only and output-input operations, though, the interaction is not driven directly by a client request. Instead, the server itself initiates message sending (and in the case of output-input operations, the server waits for a response). Implementing such operations requires a different kind of process within the server, not corresponding to any of the components in Figure 7. This process will act as an EJB client, creating a sequence of transactions and using them to invoke EJB methods. It will compute messages and dispatch them to the remote server.

We propose to treat processes like this as a third type of component. As with servlets and EJBs, instances of this component will implement a standard interface. They will be managed by a custom container. Figure 8 shows the extended architecture.

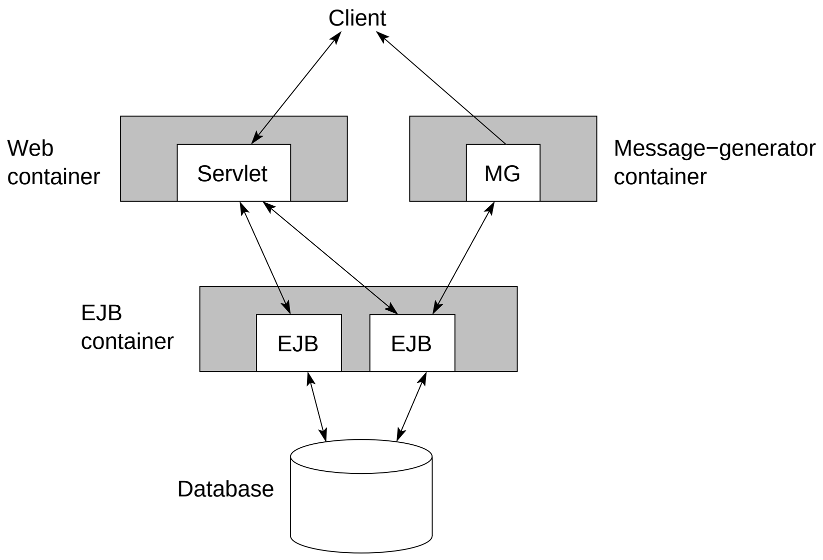

Figure 8: J2EE architecture, extended with a simple container for components that generate messages.

4.2.1 Message-generating components and JMS

J2EE specifies a standard API for “messaging”: the Java Message Service [HBS$^+$02]. JMS provides standard interfaces for such messaging abstractions as endpoints, listeners, queues, subscription, etc. It may be relevant to the basic-services architecture in general, and to the "message-generating component" in particular.

First, the JMS API is not biased toward synchronous messaging, as the servlet API is. Inbound and outbound JMS messages are not closely coupled. Replying to a message is easy, but is not required.

Second, there is a form of EJB called a "message-driven bean", which is invoked on receipt of a JMS message. In principle, messages arriving at the web service could start computation by triggering such a bean, rather than by invoking a servlet. More interestingly, message-driven beans are the only component of the J2EE architecture that is designed to run asynchronously, rather than on behalf of a client which is waiting for a response. We are investigating whether such beans could play the role suggested for message-generating components. One drawback is that message-driven beans are stateless (and there is no state object officially associated with them, comparable to the HttpSession for servlets). Another is that the only way to initiate computation with a message-driven bean is to send a message; in some cases, this may be a gratuitous extra level of indirection.

However, we are continuing to investigate the possibility of using message-driven beans. We will describe message-generating components in detail in a later deliverable.

4.3 Architectural goals

One of the goals of the ADAPT project is to implement services in such a way that they are *adaptable*. For basic services, we have chosen to focus on two forms of this problem: adaptability to system failure and adaptability to load. Our basic strategy to achieve these adaptability goals is replication.

Another major reliability goal for ADAPT basic services is that of *exactly-once execution*. In a replicated web service implementation, the computation for an operation may be repeated, in whole or in part, for several reasons. First, when a replica fails, any pending computations it was working on must be "failed over" to another replica, so they can be completed. This must work correctly even if some of their results have been saved to persistent storage. And second, since the client connects across a wide-area network, it may experience delays and failures. Thus the implementation must be prepared to receive multiple copies of any client request.

Frølund and Guerraoui [FG01] have presented important work on implementing exactly-once request semantics. The system they describe, though, is considerably simpler than J2EE. First, it is stateless. J2EE, on the other hand, has stateful components, some of which form a shared cache of rows in the database. Second, they consider requests in isolation, while for web services, we must support conversations between client and service.

4.4 Elements of replication

The J2EE model is not designed explicitly for replication. However, its component structure is actually quite suitable for replication, with only minor modifications.

The computation performed by the J2EE server can be broken down into the state of the participating components, and the invocations between them. The replication schemes we are considering (see Section 4.5) are generally *passive*. That is, there may be many replicas of a component on different hosts, but when a component is invoked, the computation is actually performed by only one replica, the "primary". Some time after the invocation completes, a message is sent from the primary to the other components, updating their state to the values held by the primary. Thus, a prerequisite for any replication scheme is an understanding of how the state of each type of component can be read and set.

For historical reasons, the state of the J2EE components is encapsulated in slightly different ways.

Component	Container	Instance key	State
Servlet	Web	Client ID	HttpSession
Session Bean	EJB	Client ID + class	Bean instance
Entity Bean	EJB	Bean key + class	Bean instance
Message generator	MG container	Unique ID	MG instance

Figure 9: Table of components in the model, with the enclosing container; the key that identifies a component instance; and the object that encapsulates component state.

Servlet The state is the HttpSession associated with a client conversation.[6] To transmit it, JBoss (and other replication schemes [Han02]) use Java serialization, on the entire HttpSession object or on the individual entries.

Stateful session bean The state is the fields of the bean instance, including private fields. (See the discussion of "passivation", in version 2.0 of the EJB specification [Sun02], section 7.4.1.)

Entity bean When the EJB container manages the mapping of the bean to the database, its state is the bean's "properties", i.e. public pairs of getter/setter methods. (See the EJB spec [Sun02], section 10.3.1.)

If a message-generator component is simply a message-driven EJB, as discussed above, then it will have no state. But if we define a new kind of component for this architecture, we will be able to specify that it is stateful. We would define the state in whatever way seems most consistent—perhaps following the model of the stateful session bean.

With each type of component, we may also speak of a key, which uniquely identifies a component instance. The containers use this key (at least logically) to look up the right component instance at invocation time. In replication, the key is used in each state update message, to match state items with components.

Servlet The instance key is the identity of the client. In a web application, this is often represented by a "cookie" sent in the HTTP transport layer, along with the messages.

Stateful session bean The key is the identity of the client, plus the bean class.

Entity bean The bean class, plus the unique key of the underlying database rows.

The key for a message-generator component will be a unique ID allocated at the time it is launched.

Figure 9 summarizes the components of the extended J2EE model, with keys and state.

From the programmer's point of view, replication imposes only a few extra conditions beyond those already specified by J2EE.

- Items stored in the HttpSession must be serializable.
- Stateful session beans are serialized and deserialized for replication, not only for passivation and activation.
- Entity beans are serialized and deserialized for replication, not only for container-managed persistence.

[6]More fully, it is the set of all objects reachable starting from the HttpSession. The same extension applies to the other state definitions.

4.5 Replication techniques

A wide variety of replication options have been studied in the literature. In general, we are considering only "passive" replication algorithms. But within this class, algorithms vary widely. In terms of communication, for example, some send a large number of messages between replicas, and some send only a minimum. In this section, we briefly examine two proposals that span this spectrum.

Huaigu Wu of the McGill group has proposed a replication scheme called "Sib". In this model, each component (servlet or EJB) is considered separately for replication. When a request arrives at the component, it is "logged" by multicasting a copy of the request to all replicas. During the course of request processing, the state of the component is periodically multicast to the replicas. When a component fails, the request can "fail over" to a replica, which can then continue the computation from the last checkpoint. In "optimistic" variants of this algorithm, request and state logging may be deferred for subtrees of the component invocation tree. This reduces the number of distinct multicast messages sent during the processing of a request; but it means that on failover, the computation must be restarted from an earlier point.

As a representative of the opposite end of the spectrum, the Bologna group has proposed a "minimalist" replication algorithm, which uses group communication as little as possible. In the course processing one request at the top (servlet) level, just one multicast is sent among the replicas in the group. This means that failover must be handled by the Web service client.

Our goal is to make possible a wide range of possible algorithms. Accordingly, we plan to develop a "replication framework" within the J2EE server code. Different replication algorithms can be implemented within this framework; in principle, they could even be swapped in and out with a configuration mechanism. In a future deliverable, we will describe this framework, and the specific replication algorithms, in detail.

5 Transaction support

This section includes material also presented in the progress report on transaction support, from the Madrid group.

5.1 Transactions within a replicated server

Within the J2EE architecture, the concept of transactions is represented by the Java Transaction API (JTA) [CM02]. This API expresses several key abstractions:

Transaction manager There is one transaction manager, which can create new transactions. It is the coordinator of the two-phase commit protocol.

Transaction Stands for a set of changes, which may be committed or aborted together. Transactions are associated with threads: there may be up to one associated with the current thread.

Transaction participant A resource holding changes that depend on a transaction. Provides prepare, commit, and abort methods for the 2PC protocol.

The most important transaction participant is the JDBC driver. J2EE requires that the transaction manager correctly handle JDBC drivers that do not support the two-phase commit protocol (i.e. do not implement the interface `javax.transaction.xa.XAResource`). The replicated database middleware used in ADAPT, though, will support 2PC, so this is not an issue.

The other major transactional data in J2EE is the state of the components, particularly entity beans and stateful session beans. This state is not directly modeled as a transactional resource, however. Instead, the EJB container treats it as a cached copy of the underlying data in the database. At key points (load, commit, and rollback), the container loads the state from the database, or flushes the bean instance from memory, in order to guarantee consistency between the cache and the database. While a bean works on a request, its data may be out of sync with the database. During this time, other threads may be blocked from accessing the bean.

J2EE does not provide for the transactional management of HttpSession state. For web service implementations, the HttpSession will probably contain nothing more than remote references to EJBs. However, it certainly can be used for more, so for consistency, we plan to add such support.

5.2 Replicated database middleware

In the architecture for BSs (see Section 4), the database is always accessed from the EJB tier. The EJBs, in turn, access the database through JDBC. Given that the application server can be replicated for availability and performance reasons, the replicated database should take into account that its clients can be replicated as well. This architecture is depicted in Figure 10.

We will consider two different replication models for the application server:

- Active replicated application server, that is, all the replicas of the application server submit the same sequence of requests to the database.
- Passive replicated application server. Only one replica of the application server, the primary, executes the requests from all clients, and accesses the database. The primary periodically checkpoints its state and multicast it to the remaining replicas, or backups. If the primary fails, another replica takes over. In case of failover, the new primary might repeat part of the work (i.e., submit duplicate JDBC operations) already performed by the failed primary.

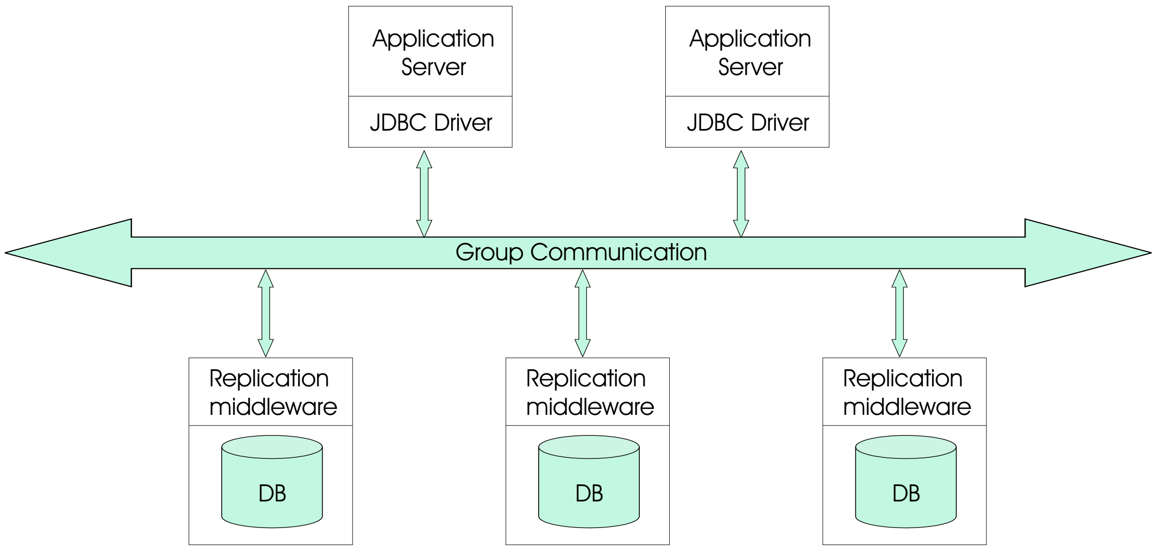

Figure 10: Database Replication Middleware

Both models require the handling of duplicate JDBC requests. In the active replication model, a mechanism will be required to identify identical requests from different replicas of the application server. On the database side, these duplicates must be detected and removed. In order to guarantee consistency, the database will require from the replicated application server to send exactly the same sequence of JDBC requests from each replica. In the passive replication model, duplicates can only happen during the fail-over. The requirement imposed by the replicated database in this case is that the replica taking over should identify duplicate requests in the same way the failed replica did. This means that independently of the assumed replication model a duplicate removal mechanism is needed.

Without loss of generality, from now on, we will assume a passive replicated application server (the active model can be seen as a special case). Some of these requests might be duplicated (during failover), but since duplicate requests are filtered out, they do not pose any inconsistency problem.

In the passive replicated approach, a transaction is only tracked by the transaction manager in the replicated application server that executes the transaction. In order to enable fail-over, it is necessary to checkpoint the transaction manager state to the rest of the replicas of the application server. The state corresponding to each transaction can be checkpointed at the end of each method invocation together with the state of the stateful session beans and the set of accessed entity beans. Clients accessing the application manager should be able to perform fail-over in a transactional context without aborting the transaction. This might require providing a modified JTA to the client.

5.3 Dynamic adaptability of databases

Our concept of dynamic adaptability for replicated databases includes different features:

- Online recovery. The ability to recover new or failed replicas of the database without stopping request processing and disrupting service as minimally as possible.
- Dynamic load balancing. The ability to balance the load at run time, without disrupting service processing.

- Dynamic control of concurrency. The ability to track dynamically the optimal level of concurrency to achieve the maximum throughput.

5.3.1 Online recovery

The goal of online recovery for replicated databases in the context of ADAPT is to introduce recovery facilities to the replicated database so that the inclusion of failed or new replicas disrupts minimally normal processing. Several protocols with a similar goal have been proposed recently [KBB01, JPA02]. Given that in ADAPT the implementation of database replication will follow a gray box approach (replication protocols are implemented on top of the database), we will use the protocols presented in [JPA02].

Our online recovery protocol basically recovers each table in a more or less independent fashion. This feature is very convenient to prevent the disruption of service processing in the recoverer replica (the one that sends the state to the new replica). Additionally, the protocol uses the logged updates for recovery instead of the database itself, what uncouples the recovery processing from the request processing. This is very beneficial since the recoverer replica can continue applying updates even to the table being recovered. Recovery is also uncoupled from the underlying group communication state transfer. This is important, since during the state transfer group communication is blocked, what would result in stopping servicing requests during this period.

Another aspect worth to mention is that of simultaneous and cascading recoveries. When recovering a set of sites in a cluster, they can restart simultaneously (very unlikely) or in a cascading fashion. In the case of simultaneous recovery the algorithm takes advantage of the group communication primitives and perform recovery of all the replicas simultaneously. Cascading recoveries are more difficult to deal with. In this case, the protocol takes advantage of the fact that recovery is performed on a per-table basis. Thereby, when a replica starts recovery, whilst a recovery is underway, it joins the recovery process in the next table that is recovered. For instance, let us assume that there is a database with 10 tables. Let us also assume that one replica has recovered tables 1-3 and it is recovering the fourth table when the new replica starts recovery. The new recovering replica will join the recovery process when the recovery of the fifth table starts and continue it till the former replica ends it. Then, the new recovering replica will continue alone the recovery of the first four tables. This approach to recovery prevents redundancies that would take place if cascading recoveries are performed independently. Additionally, the recovery process is fault-tolerant. If the recoverer replica fails, another working replica takes over the recoverer role and continues the recovery process with the existing recovering replicas.

Finally, the protocol for online recovery is adaptable in the sense that it can adapt the resources devoted to recovery according to the spare computing capacity in the replicated database. We will later discuss the importance of this issue in the context of load balancing.

5.3.2 Dynamic control of concurrency

A database, as any other software with finite resources, is able to increase its throughput with an increasing load till a threshold. Once this threshold is reached, the lack of resources yields to a trashing behavior, reducing the potential throughput. Unfortunately, this threshold changes dynamically with the workload, that is, it cannot be set at configuration time.

The proposed middleware for database replication will keep a pool of connections with the underlying database, PostgreSQL. The number of active transactions in the database can be controlled dynamically by increasing or reducing the number of connections used in parallel. The question is how to determine the optimal degree of concurrency since it is a moving target. Our plan consists in running a set of experiments to determine the database behavior under different workload conditions. From these experiments an analytical model will be synthesized. The middleware will track the behavior of the system and determine, by applying the analytical model, at which region of the throughput curve the system is. If the trashing threshold has not been reached, the degree of concurrency can be increased. Otherwise, if that threshold has been surpassed, the number of active transactions should be reduced. This means that the replication middleware will adapt itself dynamically to reach the optimal level of concurrency at each replica.

5.3.3 Load balancing

The database load balancing will be dynamic. The replication middleware should detect which replicas become overloaded and distribute their load to replicas with spare computing resources. Since each replica is dynamically adapting its optimal level of concurrency, overload cannot be detected by a trashing behavior. The number of pending requests will give an estimation of the load at a given replica. Therefore, the load balancing protocol will monitor the number of pending reques and the average response time to detect the overload of replicas. A replica whose response time increases, it is becoming overloaded, what will result in an increase of the number of pending request. The load balancing protocol will decide that some pending request at an overloaded replica can be executed at other replica to reduce the load of that overloaded replica.

Since all the request to the replication middleware are sent to all replicas, and update transactions must sent their write set to all the replicas, every replica has a precise knowledge of the behavior of the other replicas without resorting to extra messages for load balancing purposes.

6 Associated CS support

The goal of ADAPT is to support adaptable composite services (CS). We have factored the architecture into two levels: “basic services” (BS), discussed in this document, and composition, discussed in a separate deliverable [AJB+03]. However, the two levels inevitably interact. In this section, we discuss aspects of the architecture at the interface between the levels.

6.1 Service Enactment Coordinator

The architecture of composite services in ADAPT is distributed. A composite service description is compiled into process descriptions that are deployed at multiple Service Enactment Coordinator (SEC) nodes. When a composite service invokes a basic service, the actual web-service call is made by an SEC node, not by some “master” node for that CS.

Note that this architecture does not require any particular association betwen BS nodes and the invoking SEC nodes. The SEC node might be in the same local network as the BS, or the same geographical area, but remote calls are equally possible.

6.2 Sensors and Actuators

Web service definitions express the functional properties of services: the set of messages, their format and sequencing, etc. But some non-functional properties, particularly those related to performance, are also of interest to clients such as the SEC. For example, if the client is in a position to choose between two semantically equivalent services, and has access to performance information about each, it would naturally choose to invoke the one with better performance. And further, if the performance of a service can be affected by configuration, it would be advantageous to expose aspects of the configuration to clients. For example, if the throughput of a service can be configured, a client might well choose to do this before invocation, even if this meant paying a higher price.

We have discussed a number of possibilities for how such non-functional properties can be exposed and managed. We do not have a detailed proposal yet, but the basic outlines are clear.

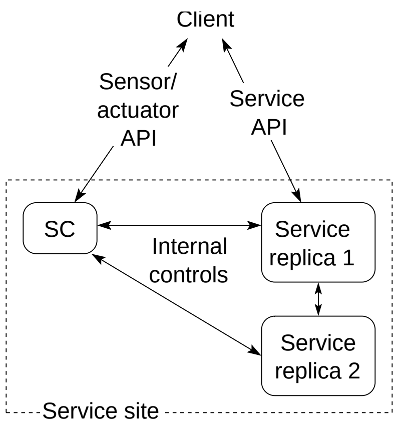

Figure 11: The service controller (SC), exposing sensors and actuators at a service site. In this configuration, it runs on a distinct host within the site.

Each service site that exposes non-functional properties in this way will include a dedicated architectural component. For now, we refer to this component as the *service controller*, or SC. The SC will be accessible to clients through a standard web-service API. It will expose abstractions which we call *sensors*, for readable properties, and *actuators*, for modifiable properties.

There are two advantages to distinguishing the SC as a separate component in the architecture. First, it makes it easier to standardize the server/actuator API, and to some extent the implementation. Second, it allows a service site to host the SC on a separate machine from the rest of the service. This would relieve the service implementation proper of the communication and computation burden of the sensors and actuators.

Note that, unlike the SEC, the SC should run on the local-area network of the service it controls. This is because we expect it will communicate with the service implementation using non-public APIs. Figure 11 shows the SC in place at a basic service site. In this example, the basic service is replicated on two machines, and the service controller runs on a third machine. Many other configurations are possible.

References

[AJB+03] Gustavo Alonso, Daniel Jönsson, Biörn Biörnstad, Simon Woodman, Stuart Wheater, and Santosh Shrivastava. CS Middleware Architecture, March 2003. ADAPT deliverable document D9.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawana. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium (W3C), March 2001. W3C Note 15. See `http://www.w3c.org/TR/wsdl/`.

[CM02] Susan Cheung and Vlada Matena. Sun Microsystems Inc. Java Transaction API (JTA), November 2002. See `http://java.sun.com/products/jta/`.

[FG01] Svend Frølund and Rachid Guerraoui. Implementing e-Transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, February 2001.

[Han02] Filip Hanik. In memory session replication in Tomcat 4. *TheServerSide.com*, April 2002. See `http://www.theserverside.com/resources/articles/Tomcat/article.html`.

[HBS+02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java Message Service version 1.1, April 2002. See `http://java.sun.com/products/jms/docs.html`.

[Jav] JavaGroups. See `http://www.javagroups.com/`.

[JBo] JBoss Group. JBoss. See `http://www.jboss.org/`.

[JPA02] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE Symp. on Reliable Distributed Systems*, pages 150–159, 2002.

[JPPM03] Ricardo Jiménez-Peris and Marta Patiño-Martínez. Transaction Support, March 2003. ADAPT deliverable document D5.

[KBB01] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*, Goteborg, Sweden, June 2001.

[LB02] Sacha Labourey and Bill Burke. *JBoss Clustering*. The JBoss Group, September 2002.

[MAMSA94] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[Sha02] Bill Shannon. *Java^{TM} 2 Platform Enterprise Edition Specification, v1.4*. Sun Microsystems, Inc., November 2002. Proposed final draft 2. See `http://java.sun.com/j2ee/1.4/docs/`.

[Sun02] Sun Microsystems, Inc. *Enterprise JavaBeans^{TM} 2.0 Specification*, 2002. See `http://java.sun.com/products/ejb/docs.html`.